Süsteemprogrammeerimine keeles C

milles räägime struktuuridest, binaarsete failide kirjutamisest, lugemisest ja stringide sisend-väljundfunktsioonidest, sellest mida tähendab const, näitame typedef käsku ja räägime sellest, mis on skoop ja loodetavasti lõpetan ma selle lause ära.

Eelmises osas (makrod)

- Makrod on defineeritud parameetriga asendused
- Lahenduvad preprotsessoris
  - #define MAX(a, b) ((a) > (b)) ? (a):(b)
- <ctype.h> lisamisel tekivad isxxxx() makrod nagu näiteks isdigit() ja islower()
- Kuna preprotsessor ei kontrolli C süntaksit, arvestada kõrvalefektidega (kui makro kasutab oma argumenti mitu korda) ja sellega, et tehtejärjekorda sunnitakse rohke sulukasutusega.
- Eelisteks eri tüüpi argumendid ja vähene *overhead*.

Eelmises osas (enum)

- Enumeration defineerib grupeeritud konstante:

```
enum mode { STANDBY, INIT, RUNNING, IDLE, ERROR = 255; };
enum mode currentmode, nextmode;
currentmode = STANDBY;
...
if (currentmode != ERROR) do_happy_dance();
else regenerate_limbs();
```

- Eelis kompilaatori tüübikontrolli osas

Eelmises osas (switch, break, continue)

- Switch:

```
switch (mode) {
case INIT:
case IDLE:
moan(); break;

case RUNNING:
run_like_hell(); break;

default:
do_happy_dance();
}
```

- break – väljub parajasti aktiivsest tsüklist või switch-blokist
- continue – alustab for, while, do..while puhul järgmist tsüklit

Eelmises osas (Preprotsessori käsud)

- Preprotsessor töötab enne kompileerimist
  - #include – lisab *header* faili (või suvalise faili)
  - #define – defineerib makro või konstandi
  - #ifndef, #ifdef, #endif – tingimuslik lisamine
  - #if, #else, #elif, #endif
  - #undef – tühistab definitsiooni
- *Headeri puhul levinud taktika:*

```
#ifndef _HEADER_FILE_
#define _HEADER_FILE_
   ... // sisu siia
#endif
```

Eelmises loengus (Pointerid)

Pointeris hoitakse mäluaadressi

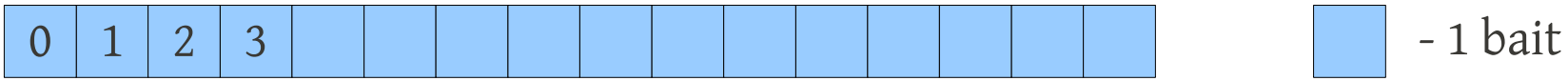
Objekti mäluaadressi saab operaatoriga &

Kui pointeris on aadress, võtab operaator * selle aadressi poolt viidatud sisu

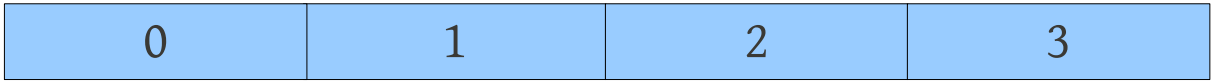
Eelmises loengus (tüübid ja mahud)

Erinevad tüübid võtavad mälu erinevalt:

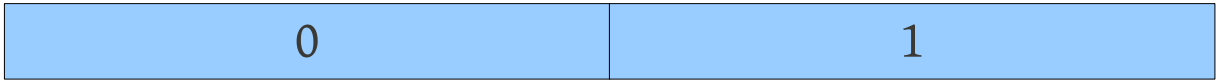
```
char c[N]; char *pc = c[0]; // *(pc+1) == c[1];
```




```
int i[N]; int *pi = i[0]; // *(pi+1) == i[1];
```




```
double d[N]; char *pd = d[0]; // *(pd+1) == d[1];
```



Kui palju võtab mälu pointer?

- sõltuvalt arhitektuurist

Eelmises osas (pointerite võrdlemine)

Pointereid ei ole mõtet võrrelda

Erand: NULL olemise kontroll

Erand 2: Sama massiivi elemendid on järjest

Eelmises osas (pointer vs massiiv)

Pointeri poole saab pöörduda nagu massiivi poole

```
int *ptr; ptr=intimassiiv; ptr[3]=4; // või *(ptr+3)=4;
```

Massiivi poole saab pöörduda nagu pointeri poole

```
int intimassiiv[4]; *(intimassiiv+3)=4; // intimassiiv[3]
```

Erinevus: Massiivi aadressi ei saa vahetada

```
intimassiiv = intimassiiv+3; // VIGA!
```

Eelmises osas (Massiivid funktsiooniargumentides)

Funktsiooni käivitamisel tehakse argumentidest koopiad

Reeglina tähendab see, et funktsioon ei saa oma argumente muuta

Funktsioon saab muuta argumendiks olevat massiivi, sest argumendiks on aadress, mitte massiiv ise.

Eelmises osas (dünaamiline mäluhaldus)

Mälu andmiseks on funktsioonid, mis tagastavad pointeri hõivatud mälule

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void * realloc(void *ptr, size_t size);
```

Võetud mälu tuleb vabastada

```
void free(void *ptr);
```

Eelmises osas (Pointer funktsioonile)

Funktsioonidele on võimalik anda argumentideks teisi funktsioone.

```
int runIntFunction( int (*func)(int), int arg ) {
    (*func)(arg);
}
i = runIntFunction(add1, i);
```

Tänases loengus

- Massiivid
- Struktuurid
- Failidest lugemine ja neisse kirjutamine
- Ajutised failid
- Stringid: lugemine, kirjutamine, müramine
- const
- typedef
- Skoop

Ühemõõtmelised massiivid

- Fikseeritud (*stack*is) ja dünaamilised (*heap*is) massiivid käituvad täpselt ühtmoodi, kui deklareerimine välja arvata.

```
int vec[100]; /* staatiline */
```

```
int *vec;    /* dünaamiline */
vec = (int*)malloc(sizeof(int)*100);
```

```
/* kasutamine */
vec[70] = 1; /* või */ *(vec+70) = 1 ;
```

```
/* initsialiseerimine */
for (i=0;i<100;i++) vec[i]=0; /* ei ole tõhus */

int *ptr = vec; int *end = vec+99; *end = 0;
while(ptr != end) *ptr++ = 0;
```

```
/* funktsiooni prototüüp */
void func(int * ptr);  /* või */
void func(int ptr[]);
```

Fikseeritud kahemõõtmelised massiivid

- ***Stack*is olevad massiivid**
  Hõivamine: int fixed[50][100];
- **Ligipääs:** fixed[5][9] = 1; /* või */
  fixed[0][5*100+9] = 1; /* või */
  fixed[1][4*100+9] = 1; /* jne */
- **Initsialiseerimine:**

  for(i=0;i<50;i++) for(j=0;j<50;j++) fixed[i][j] = 0; /* ebatõhus */

  int *ptr = fixed[0]; int *end = fixed[49]+99; *end = 0;
  while(ptr != end) *ptr++=0;
- **Funktsioonile andmine:**
  Prototüüp: void func(int fixed[50][100]);

Dünaamilised kahemõõtmelised massiivid

- Asuvad *heap*is – see on paindlikum

Hõivamine

```
int **dynamic;
dynamic = (int**)malloc(sizeof(int*)*50);
dynamic[0] = (int*)malloc(sizeof(int)*50*100);
for (i=1;i<50;i++) dynamic[i]=dynamic[i-1]+100;
```

Ligipääs

```
dynamic[5][9] = 1; /* või */
dynamic[0][5*100+9] = 1;      /* või */
dynamic[1][4*100+9] = 1;      /* jne... */
```

Initsialiseerimine

```
int *ptr = dynamic[0];
int *end = dynamic[49] + 99; *end = 0;
while (ptr !=end) *ptr++=0;
```

Prototüüp

```
func(int** vec);
```

Massiividest veel

- Massiivid tuleb initsialiseerida, nad ei ole vaikimisi 0-dega täidetud
- Massiivi mälu tuleb vabastada, seda tehakse vastupidiselt hõivamisele:

```
free(dynamic[0]);
free(dynamic);
```

- Dünaamiliselt hõivatud massiivi "dynamic" indekseerimisel kujul dynamic[i][j], pole vaja korrutada.
- Dünaamiline 3d massiiv on pointerite massiiv, mis viitab dünaamilistele 2d massiividele.

Meeldetuletus teemal *casting*

- Tüübiteisendused sunnivad tehete tulemusi olema mingites kindlates tüüpides (teisendamine vägisi)

```
int i = 2; double d = 1.7;
i * d;  /* tüüp double, väärtus 3.4 */
(int)(i*d) /* tüüp int, väärtus 3 */
i*(int)d /* tüüp int, väärtus 3 */
```

- Sunnime funktsiooniargumendi õigesse tüüpi:

```
d = sqrt((double)i);
```

Pointerite teisendamine

- Mõnikord tekib situatsioon, kus peame defineerima pointeri teadmata mis tüübile ta viitab
- Selliseks üldiseks pointeritüübiks on void tüüpi pointer. void tüüp teiseneb vajadusel ise.

```
int a[10]; void *pv = &a[1]; int *pi=&a[2];
int i = pi - pv; /* hoiatab, kuid saab hakkama */

int i; double d,e;
void *pv0=&i, *pv1 = &d;
e = *pv0 + *pv1;      /* ei kompileeru */
e = *((int*)pv0)+*((double*)pv1); /* töötab */
```

Struktuurid

- Süntaks **`struct nimi{väljad};`**
- Struktuur on liittüüp, mis koosneb teistest pisematest tüüpidest
- Struktuuri komponente on võimalik lugeda
- Peaaegu kõik, mis on lubatud sisseehitatud funktsioonidele, on lubatud ka tüüpidele
    - Lubatud: Massiivid struktuuridest, struktuuride tagastamine
    - Võimatu: loogikaoperaatorite ja aritmeetika defineerimine
- Struktuuri alamkomponendina struktuuri kasutamine on lubatud.

Struktuuri kasutamine

- Defineerimine:

```
struct person {
   char name[20]; char last_name[30];
   struct {
       char street[30]; char city[30];
       unsigned int house, zip;
   } address;
   struct person *mother, *father;   // rekursioon!
};
```

- Kasutamine:

```
int check_relation (struct person person1, struct
   person person2);

typedef struct person ID;
ID myself;                  // deklareerime muutuja
strcpy (myself.name,"Manivalde");
```

Pointer struktuurile

- Andmetele ligipääsuks tuleb kasutada "." operaatorit. Näiteks: myself.name
- Kui struktuurile on pointer, kasutame "->" operaatorit.

```
struct person *p1;
p1 = (struct person*)malloc(sizeof(struct person));
p1 -> address.house = 48;
(*p1).address.house = 48;
free (p1);

strcpy (person1->father->last_name, person1->last_name);
```

"." ja "->" on kõige siduvamad tehted ja assotsieeruvad paremalt vasakule

Massiivid struktuuridest

- Struktuuridest saab teha nii fikseeritud kui dünaamilisi massiive:

```
ID fixed_family[20];
ID *dynamic_family;
dynamic_family = (ID*)malloc(sizeof(ID)*20);
```

- Järgmised read on võrdväärsed

```
strcpy(fixed_family[10].name, “Manivalde”);
strcpy(dynamic_family[10].name, “Manivalde”);
```

Rekursiivsed struktuurid

- Lingitud listi näitel

```
struct ListElem {
  int number;
  struct ListElem *next;
}
```

- Binaarse puu näitel

```
struct TreeNode {
  void *data;
  struct TreeNode *left;
   struct TreeNode *right;
}
```

Boonus

Struktuuri mäluhõivest

Binary I/O - fread()

- size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)

```
FILE      *fp;
char      x[500];
if ((fp=fopen("pic.jpg, "r")) == NULL) {
  fprintf (stderr, "Can't open file\n");
  exit(EXIT_FAILURE);
}
fread(x, sizeof(x[0]), 10, fp);
fclose(fp);
```

Binary I/O - fwrite()

- size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream)

```
FILE      *fp;
char flag = '1';
int size = 3456;

if ((fp=fopen("output", "w")) == NULL) {
  fprintf (stderr, "Can't open output file\n");
  exit(1);
}
fwrite(&flag, sizeof(char), 1, fp);
fwrite(&size, sizeof(int), 1, fp);
…..
fclose(fp);
```

Ajutised failid

- FILE* tmpfile();
- Tagastab pointeri *stream*-ile, mis on avatud nii lugemiseks kui kirjutamiseks
- Ajutine fail kustutatakse pärast selle sulgemist fclose() funktsiooniga või kuni programmist väljudes. Mõnes süsteemis hiljem (taaskäivitamisel).
- Faili asupaik sõltub süsteemi ja standardteegi valikutest.

Binary I/O - Lisavõimalused

- int fflush(FILE *fp); // kirjutab failipuhvri tühjaks
- void setvbuf(FILE *fp, char *buf, int mode, size_t n);
  - mode: _IOFBF, _IOLBF, _IONBF (fully-, line-, not-)
- Positsioon failis:
  - int fseek(FILE *fp, long offset, int place);
    - kus koht = {SEEK_SET, SEEK_CUR, SEEK_END}
  - long ftell(FILE *fp);
  - void rewind(FILE *fp); // == (void)fseek(fp, 0L, SEEK_SET)

  - int fgetpos(FILE *fp, fpos_t *pos); // pos võib olla rohkem kui long!
  - int fsetpos(FILE *fp, const fpos_t *pos);

Stringide lugemine (gets, fgets)

- char *gets(char *s);
  - gets loeb standardsisendist stringi.
  - reavahetus asendub \0 märgiga
  - **ära kunagi kasuta!** sa ei saa anda ette puhvri suurust
- char *fgets(char *s, int size, FILE *fp);
  - fgets loeb viidatud puhvrisse stringi failist kuni reavahetuse või EOF-ni. Maksimaalselt *size* tähemärki.
  - reavahetus ei asendu, \0 lisatakse
- EOF puhul tagastub NULL

Stringide kirjutamine (puts, fputs)

- int puts(const char *s);
  - kirjutab stringi standardväljundisse
  - kirjutab \0 asemel reavahetuse \n
  - **ära kunagi valjusti ütle!**
- int fputs(const char *s, FILE *fp);
  - kirjutab stringi viidatud *stream*'i
  - reavahetust \n ega \0 märki ei lisata
- Edu korral tagastub mittenegatiivne arv, vea korral EOF

sprintf(), sscanf()

- Stringide vormindatud lugemine ja kirjutamine

```
int sprintf(char *s, const char *format, ...)
int sscanf(const char *s, const char *format, ...)
```

- fprintf ja fscanf stringiversioonid

```
char str1[]="1 2 3 go", str2[100], tmp[100];
int a,b,c;
sscanf(str1, "%d%d%d%s", &a, &b, &c, tmp); // reads
from its first arg
sprintf(str2, "%s %s %d %d %d\n", tmp, tmp, a, b, c);
printf("%s", str2);
```

```
go go 1 2 3
```

string.h

- Üldinfo
  - stringilibra funktsioonid arvestavad, et \0 lõpetab stringi
  - kui \0 puudub, võib väljund olla "natuke" vale

```
char dst[100], s[]="abc";
s[3] = 'd'; /* '\0' kirjutatakse üle */
strcpy(dst, s); /* oih! */
```

Stringi pikkus

- size_t strlen(const char *s);
  - tagastab stringi pikkuse
  - const sellises deklaratsioonis sümboliseerib, et s ei ole funktsioonisiseselt muudetav
  - size_t on enamasti kas unsigned int või unsigned long

```
size_t strlen(const char *s) {
 size_t n;
 for (n = 0; *s != '\0'; s++)
    n++;
 return n;
}
```

```
size_t strlen(const char *s) {
    const char *t = s;
    while (*t)
       t++;
    return t - s;
}
```

Stringi kopeerimine

- char *strcpy(char *dest, const char *src);
  - kopeerib stringi *src* stringiks *dest*
  - jälgida, et *dest* oleks *src* mahutamiseks piisavalt suur
- char *strncpy(char *dest, const char *src, size_t n);
  - kopeeritakse maksimaalselt n baiti
  - kui on täpselt n, siis \0 jääb lõpust ära
  - kui on vähem kui n, kirjutatakse ülejäänutesse \0

Stringide võrdlemine

- int strcmp(const char *s1, const char *s2);
  - võrdleb stringe omavahel
  - tagastab 0, kui stringid on võrdsed
  - kui s1 > s2, siis tagastab arvu , mis on suurem kui 0
  - ja vastupidi

```
"str1"   >  "str0"
"str1"   == "str1"
"str1"   <  "str2"
```

  - levinud viga:

```
if ( strcmp(a, b) ) ... // VIGA!
```

- int strncmp(const char *s1, const char *s2, size_t n)

Stringide jupitamine

- char *strtok(char *str, const char *delim);
  - tagastab stringi *str* järgmise jupi või NULL, kui neid enam pole
  - argumendina esimeses väljakutses jagatav string, edaspidi NULL
  - delim on string juppe eraldavatest charidest
  - puudusteks on see, et me ei saa teada, mis see eraldusmärk oli ja see, et esimest argumenti muudetakse

strtok() näide

```
void print_tokens(char *line)
{
  char whitespace[]=" \t\f\r\v\n";
  char *token;
  for (token = strtok(line, whitespace); token != NULL;
          token=strtok(NULL, whitespace)) // NULL!
   printf("Järgmine on %s\n", token);
}
```

Ülevaade stringifunktsioonidest (1)

- **char *strcat(s, cs)** Liidab stringi s otsa stringi cs ja tagastab s-i
- **char *strncat(s,cs, n)** Lisab stringi s otsa maksimaalselt n stringi cs märki
- **char *strcpy(s, cs)** Kopeerib cs-i s-iks, ka \0
- **char *strncpy(s,cs)** Kopeerib cs-i maksimaalselt n tähemärki s-iks, ülejäänud määrab \0-deks
- **char *strtok(s,cs)** Leiab stringi s seest cs-i märkidega eraldatud osad
- **size_t strlen(cs)** Tagastab cs-i pikkuse (v.a. \0)
- **int strcmp(cs1,cs2)** Võrdleb cs1 ja cs2, tagastab nullist suurema, võrdse või väiksema arvu vastavalt sellele, kas cs1 on cs2-st vastavalt suurem, võrde või väiksem
- **int strncmp(cs1, cs2, n)** Võrdleb cs1 ja cs2 esimesi n-i tähte omavahel

Ülevaade stringifunktsioonidest (2)

- **char *strchr(cs,c)** Annab esimese c esinemise peale stringis s pointeri
- **char strrchr(cs,c)** Annab viimase c esinemise peale stringis s pointeri
- **char *strpbrk(cs1, cs2)** Otsib stringist cs1 esimese cs2 tähe esinemise ja annab sellele pointeri
  - nt: strpbrk(“Selle lause lõpp on kirjas maja peal aadressil”, “cba”) annab tagasi pointeri tähele a sõnas "lause"..
- **char strstr(cs1, cs2)** Otsib stringist cs1 alamstringi cs2
  - Eelmised 4 annavad ebaedu korral kõik tagasi NULL
- **size_t strspn(cs1, cs2)** Tagastab cs1 algusest pikkuse, mis koosneb ainult cs2-s olevatest tähtedest.
  - nt:. strspn("Siil udus", “ilu S”) == 6
- **size_t strcspn(cs1, cs2)** Tagastab cs1 algusest pikkuse, milles ei esine cs2 tähti
  - nt strcspn("Siil udus", "abs") == 8

string.h failist leiate neid veel!

strpbrk()

- char *strpbrk(const char *s, const char *accept);
  - pbrk – pointer to break

```
char string[100] = "The 3 men and 2 boys ate 5 pigs\n";
 char *result;
 printf( "1: %s\n", string );
 result = strpbrk( string, "0123456789" );
 printf( "2: %s\n", result++ );
 result = strpbrk( result, "0123456789" );
 printf( "3: %s\n", result++ );
 result = strpbrk( result, "0123456789" );
 printf( "4: %s\n", result );
```

strspn()

- size_t strspn(const char *s, const char *accept);
  - ütleb mitu tähte stringist koosneb täielikult *accept*-i tähtedest

```
char string[] = "cabbage";
int  result;
result = strspn( string, "abc" );
printf( "The portion of '%s' containing only "
 "a, b, or c is %d bytes long\n", string, result );
```

strstr()

- char *strstr(const char *haystack,
  const char *needle);

```
char str[] =    "lazy";
char string[] = "The quick brown dog jumps over the lazy fox";
char fmt1[] =   "         1         2         3         4         5";
char fmt2[] =   "12345678901234567890123456789012345678901234567890";
char *pdest;    int  result;

printf( "String to be searched:\n\t%s\n", string );
printf( "\t%s\n\t%s\n\n", fmt1, fmt2 );
pdest = strstr( string, str );
result = pdest - string + 1;
if( pdest != NULL )
   printf( "%s found at position %d\n\n", str, result );
else
   printf( "%s not found\n", str );
}
```

const tüübid

- const tähistab, et pärast initsialiseerumist antud muutujat ei muudeta
  - millal on parem kui enum või makro?
    - väärtus selgub programmi jooksmise käigus
    - kasutatakse kohas, kus on vaja & aadressioperaatorit
    - kompilaator/debugger peab märkama
    - püüdes sundida funktsiooni mitte muutma oma argumente, näiteks massiivi elemente

```
int foo(const int arr[], int exp) {
...
arr[j] = exp; // VIGA SIIN
```

Tähelepanek: const char *s ütleb, et s poolt viidatud baite ei tohi muuta. Pointerit ennast võime me muuta! Kui tahame muutumatut pointerit, deklareerime char *const s.

Kõik, mulle aitab!

Loeng on läbi